MARL: A Multi-Agent Reinforcement Learning Framework for Buffer Management in Multi-Tenant Cloud Databases

Xin Huang, Yu-Ang Zhang, Hongtao Wang, Peiquan Jin
University of Science and Technology of China, Hefei, China

***Abstract*—In modern Cloud Database-as-a-Service (DBaaS), multi-tenancy is essential for maximizing hardware utilization and reducing operational costs. However, managing the shared buffer pool remains a major challenge. Traditional buffer management schemes, such as global eviction or static partitioning, lack the agility to handle the highly dynamic and unpredictable workloads that are typical in industrial cloud environments, often leading to severe Service Level Agreement (SLA) violations or resource fragmentation. In this paper, we present MARL (Multi-Agent Reinforcement Learning), an industrial-grade buffer management framework for multi-tenant cloud databases. MARL models each tenant as an independent RL agent that learns to adjust its buffer allocation in a cooperative manner. We propose a specialized SLA-aware cost model that translates hit-ratio degradation into economic penalties, enabling the system to minimize overall SLA violations through a decentralized "*Grow*/*Shrink*" mechanism. To demonstrate its practicality, we implemented MARL within `OpenGauss`, a production-ready open-source database engine. Our implementation ensures that the RL inference and fine-tuning processes are entirely off the critical path, incurring negligible runtime overhead. Experimental results on diverse workloads show that MARL outperforms state-of-the-art heuristic and static methods, reducing the average SLA cost by up to 26.3% and the buffer miss ratio by up to 11.7%. In simulated high-latency cloud storage scenarios, our approach improves throughput by up to 72.2% and reduces average latency by 45.1%. These results confirm that MARL provides a scalable, robust, and cost-effective solution for resource management in large-scale multi-tenant cloud database systems.**

***Index Terms*—Cloud service, Multi-tenant database, Buffer management, Reinforcement learning**

I. Introduction

Over the past decade, relational Database-as-a-Service (DBaaS) [1] has been widely adopted by enterprises for cloud database services. DBaaS providers such as Amazon Aurora, Microsoft Azure SQL Database, and Google Cloud SQL are primarily designed to support Online Transaction Processing (OLTP) and Hybrid Transaction/Analytical Processing (HTAP) workloads. Typically, cloud database services are multi-tenant [2]–[5], meaning that multiple databases from various customers share the same physical data center resources [6]. It is a significant challenge for cloud service providers to reduce costs [7], [8]. If providers allocate the maximum resources for each tenant, they can fully satisfy performance demands. However, this strategy is not cost-efficient. Consequently, DBaaS providers often seek to oversubscribe resources, offering tenants more resources than are physically available on the underlying infrastructure.

In a multi-tenant resource-sharing model, tenants typically use fewer resources than initially promised, allowing service providers to oversubscribe without immediate consequences. Increasing the degree of oversubscription can significantly reduce operational costs for service providers [9]. However, this approach may impact tenant performance due to potential resource shortages. Despite this, oversubscription remains a viable strategy. By dynamically reallocating resources to tenants with higher demands, DBaaS providers can mitigate the negative effects of resource shortages, thereby minimizing performance degradation.

In this paper, we study the problem of managing shared buffer pool resources in a multi-tenant cloud database. The buffer pool serves as a cache for database pages, reducing the number of direct disk accesses. The allocation of buffer pool resources is critical to tenant performance. Traditional buffer management techniques focus on improving hit ratio without considering the tenant's SLA (Service Level Agreement) [10]. Global replacement algorithms (e.g., LRU and ARC) manage all pages in a single pool, without isolating tenants. While static partitioning ensures isolation, it results in significant resource waste when workloads change.

To address the shared-buffer management in multi-tenant cloud databases, this paper proposes a novel approach based on multi-agent reinforcement learning to improve buffer management efficiency in resource-sharing cloud environments. Briefly, this paper makes the following contributions:

(1) We propose **MARL**, a multi-agent RL framework for multi-tenant buffer management. Each tenant is modeled as an independent agent that executes *Grow* or *Shrink* actions. An allocator coordinates these actions to redistribute buffer from under-utilized tenants to those experiencing buffer resource pressure.

(2) We design a multi-agent cooperation mechanism based on the proposed SLA cost model. In our RL design, agents will release buffer space if doing so significantly reduces the global SLA cost without causing unacceptable local performance penalties. We leverage Proximal Policy Optimization (PPO) to ensure stable and efficient learning.

(3) We implement MARL in `openGauss` and perform extensive evaluations against various baselines. Experiments show that MARL effectively minimizes miss ratios and SLA costs. Under a simulated cloud environment with high cache-miss latency, MARL achieves substantial improvements in both throughput and tail latency. Further, we demonstrate that the RL model scales well with the number of tenants, and its inference and fine-tuning overhead is lightweight and completely off the critical path.

II. Related Work

Page replacement algorithms determine which page to evict when the buffer is full. Traditional replacement algorithms (e.g., LRU, LFU, and ARC [11]) balance recency and frequency but often fail to capture complex workload distributions. Recently, machine learning-based policies—such as LeCaR [12], LRB [13], GLCache [14], and HALP [15]—have been proposed to optimize eviction decisions by leveraging historical access patterns and addressing training overheads. However, these optimization techniques do not distinguish among tenants and do not organize pages from different tenants into a single cache structure, so these schemes cannot be applied directly to solve the multi-tenant buffer management problem.

LAMA [16] performs a dynamic programming algorithm to obtain the optimal buffer allocations, achieving the lowest costs assessed by the Miss Ratio Curve (MRC). OSCA [17] is similar to LAMA. The difference is that LAMA uses miss ratios or average response time as costs, while OSCA aims to maximize the hit traffic. Additionally, OSCA proposes a new metric, the re-access ratio (RAR), to quantify reuse distance and to construct MRCs based on it. Cliffhanger [18] uses a shadow queue to capture the performance cliffs (multiple hits occur in the shadow queue) and allocates more cache space to the tenant whose shadow queue observes the performance cliffs. While effective for Content Delivery Networks (CDNs), the substantial computational overhead of generating and maintaining MRCs makes these approaches difficult to deploy in high-performance DBaaS environments.

SAM [19] is a stability-aware buffer pool manager designed for multi-tenant embedded databases, motivated by the observation that reactive adaptation often leads to performance instability under contention. It employs the AURA policy, which combines historical efficiency (H-factor) with forward-looking marginal gain (V-factor), and adopts a Two-Pool model to separate resource guarantees from opportunistic optimization. However, SAM relies on heuristic control, which limits resource utilization. Moreover, SAM primarily targets embedded database scenarios and prioritizes allocation stability over global performance optimality.

III. Design of MARL

In this section, we detail the design of MARL. We first describe the SLA metric in Section III-A. Then, in Section III-B, we present the SLA cost model. Section III-C discusses the architecture of MARL. Section III-D presents the details of RL agents. And finally, Section III-E shows the specific buffer reallocation process of the allocator.

A. Measure of SLA

Service Level Agreement (SLA) [10] defines the minimum performance guaranteed by a cloud service provider to a tenant. In shared buffer management, the SLA is primarily measured by the hit ratio. We define a baseline hit ratio, $HR_{promised}$, which represents the expected performance when a tenant is statically allocated their promised buffer capacity under the default buffer replacement policy. If the actual hit ratio, HR_{actual}, falls below this baseline, an SLA violation occurs. To penalize such violations, we define the Hit Ratio Degradation (HRD) as the performance violation, formulated as Eq. 1:

$$HRD = \max(HR_{promised} - HR_{actual}, 0) \tag{1}$$

$HR_{promised}$ can be measured in real time by maintaining an additional queue structure with the default replacement policy and a capacity equal to the promised buffer size.

B. SLA-based Cost Model

The SLA cost model quantifies the penalty incurred by the hit-ratio degradation of tenants, defined as Eq. 2:

$$SLA\ Cost = \sum_{i=1}^{N} m_i \cdot HRD_i \tag{2}$$

Here, m_i denotes the SLA price that tenant i is willing to pay when its performance requirements are completely met, and N is the total number of tenants. Our algorithm dynamically reallocates tenant buffer sizes to minimize the total SLA cost across all tenants.

C. Architecture of MARL

Fig. 1 shows the overall structure of MARL, which consists of three main components: (1) *Monitor*, (2) *Agent Layer*, (3) *Allocator*.

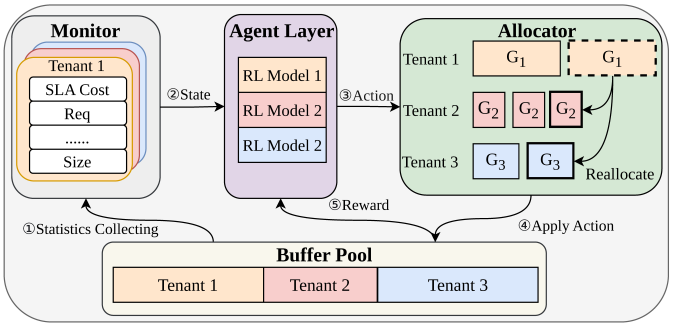


Fig. 1: Architecture of MARL.

Monitor. Monitor continuously collects runtime statistics from the database system for each tenant. These statistics include the SLA cost, the actual buffer size, the promised buffer size, and the request traffic intensity. The collected metrics are aggregated over a fixed control interval and normalized to construct the state representation for each RL agent. By providing both tenant-level and system-wide statistics (e.g., average SLA cost and average promised-to-actual buffer ratio), the monitor enables agents to reason about their local performance in the context of global system conditions.

Agent Layer. Each tenant is modeled as an RL agent that observes its own state and decides an action at each decision epoch independently. Each tenant's action consists of *Grow* and *Shrink*, which indicate whether the tenant intends

to acquire additional buffer space or release part of its currently allocated buffer. This binary decision design simplifies learning and reduces the risk of unstable behavior, while still allowing fine-grained control through repeated small buffer adjustments. All agents share the same policy network structure but maintain separate observations and actions, enabling scalable learning across many tenants.

Allocator. The allocator aggregates the actions generated by the agent layer across all tenants. Specifically, it calculates the total buffer space released by tenants who decide to *Shrink* and redistributes it to those requesting an *Grow*. Finally, the allocator generates a reallocation action for the buffer pool to execute.

D. Design of RL Agents

MARL formulates the multi-tenant buffer management into a multi-agent RL problem. It defines the RL states, actions, and reward for each agent.

RL States. The state S_t^i of the RL agent for tenant i at time t is defined in Eq. (3):

$$S_t^i = (Avg\_SLA_t, SLA_t^i, Avg\_RP_t, RP_t^i, Size_t^i, Req_t^i) \quad (3)$$

Here, Avg_SLA_t and SLA_t^i denote the average SLA cost of all tenants and the specific SLA cost of tenant i, respectively. Similarly, Avg_RP_t represents the average ratio of promised buffer size to actual buffer size across all tenants, while RP_t^i indicates this ratio for tenant i. Finally, $Size_t^i$ is tenant i's proportion of the total buffer size, and Req_t^i represents its proportion of buffer requests.

The RL state integrates the tenant's individual resource access status and utilization with the system's average resource usage. By leveraging this information, the RL agent can effectively determine whether the current buffer allocation is insufficient or excessive.

RL Actions. Each agent chooses one discrete action at each decision epoch t. MARL defines two actions for each agent to manage the tenant buffer: *Shrink* and *Grow*. Action *Shrink* is to decrease the buffer size of the tenant by a size unit G_i while *Grow* will increase the buffer size from the free buffer release by other tenants who decide to decrease the buffer size.

The buffer adjustment size unit G_i is defined as Eq. (4).

$$G_i = \frac{\sum Actual\ buffer\ size}{\sum Promised\ buffer\ size} * Promised_i * 3\% \quad (4)$$

Here, $Promised_i$ refers to the promised buffer size of tenant i, $\sum Actual\ buffer\ size$ represents the total actual buffer size of all tenants, and $\sum Promised\ buffer\ size$ represents the total promised buffer size of all tenants. The unit size is to avoid aggressive or negligible buffer resource adjustments.

RL Reward. The reward design directly reflects the system's optimization objective: minimizing overall SLA cost.

When tenant i chooses to *Shrink*, its released buffer is distributed to other tenants (e.g., tenant j). The reward for tenant i accounts for its own reduction in SLA cost and its contribution to reducing others' SLA costs. The reward function is defined as Eq. 5:

$$reward_i = -\left(\Delta SLA\ Cost_i + \sum_j R_j \cdot \Delta SLA\ Cost_j\right) \quad (5)$$

Here, $\Delta SLA\ Cost$ represents the change in SLA cost after the adjustment. R_j represents the proportion of resources tenant j received from tenant i relative to the total resources tenant j obtained. This reward function encourages agents to release the buffer if doing so significantly benefits other tenants without severely hurting their own performance. The logic applies symmetrically to tenants choosing to *Grow*.

The $\Delta SLA\ Cost$ of all tenants will be calculated in a fixed time period after their buffer adjustment is done. The period length is set to 4s in our design.

Algorithm 1: Buffer Reallocation

Input : Set of tenants $\mathcal{T}$; Current actual buffer size $B = \{B_1, B_2, \ldots, B_{|\mathcal{T}|}\}$; Promised buffer size $P = \{P_1, P_2, \ldots, P_{|\mathcal{T}|}\}$; Actions from Agent Layer $A = \{a_1, a_2, \ldots, a_{|\mathcal{T}|}\}, a_i \in \{\text{Grow, Shrink}\}$
Output: Updated actual buffer size B^{new}; Resource transfer matrix $\mathcal{R}$ (used for calculating R_j in Reward)

```
// Phase 1: Calculate Granularity and Partition Tenants
B_total ← Σ_{k∈T} B_k;
P_total ← Σ_{k∈T} P_k;
Initialize queues Q_shrink ← ∅, Q_grow ← ∅;
Initialize tracking arrays V_avail ← {0}, V_need ← {0};
Initialize transfer matrix R_{|T|×|T|} ← 0;
for each tenant i ∈ T do
    G_i ← (B_total / P_total) × P_i × 3%;
    if a_i == Shrink then
        Push i into Q_shrink;
        V_avail[i] ← G_i ;      // Buffer size to give up
        B_i ← B_i − G_i ;               // Release buffer
    else
        Push i into Q_grow;
        V_need[i] ← G_i ;        // Buffer size required
// Phase 2: Reallocation Matching Process
while Q_grow is not empty and Q_shrink is not empty do
    j ← Front(Q_grow) ;                  // Current Grower
    i ← Front(Q_shrink) ;              // Current Shrinker
    // Determine the transferable amount
    Δ ← min(V_need[j], V_avail[i]);
    // Allocate resource and update record
    B_j ← B_j + Δ;
    V_need[j] ← V_need[j] − Δ;
    V_avail[i] ← V_avail[i] − Δ;
    R_{i,j} ← R_{i,j} + Δ ;  // Record that j gets Δ from i
    // Check if the current tenant's resource need is fulfilled
    if V_need[j] == 0 then
        Pop Q_grow ;  // Grower j is fully satisfied
    if V_avail[i] == 0 then
        Pop Q_shrink ;      // Shrinker i is exhausted
return B, R;
```

E. Buffer Reallocation

The allocator's buffer reallocation process is detailed in Algorithm 1. Instead of processing each action individually,

the allocator processes them in batches. At each decision interval, the allocator computes the action for each tenant, collects the buffer resources released by agents who choose to *Shrink*, and redistributes them to agents who request to *Grow*. Resources are distributed until the released buffer is completely exhausted. To ensure exploration and fairness during training, the order of distribution is randomized. However, in a production environment, this approach can be readily adapted to a priority-based queue, with high-priority tenants served first.

IV. Performance Evaluation

In this section, we compare our proposed MARL with several existing buffer management policies, including SAM, static allocation, and equal allocation.

A. Experiment Setting

All experiments are conducted on a server equipped with an Intel(R) Xeon(R) Gold 6348 CPU (2.60 GHz, 28 cores), 256 GB of 3200 MT/s DDR4 memory, and a Western Digital 4 TB BLACK AN1500 PCIe SSD. The server runs Ubuntu 22.04.01 LTS. Models are pre-trained in a Python simulator and evaluated in `OpenGauss` 5.1.0 [20].

Implementation Details. We implemented MARL in the open-source relational database `OpenGauss`[1]. We identify different tenant requests based on the database name specified in each access request. Specifically, we introduced a dedicated background thread that periodically collects tenant metrics and transmits the state information to the RL agent via shared memory. Once the allocator computes the reallocation action, the results are sent via shared memory to the background thread, which then resizes the corresponding tenants' buffer pools. After the reward is computed and returned to the RL agents, we repeat the process until the overall SLA cost stabilizes. Notably, this entire adjustment process operates completely off the critical path of buffer requests.

We develop our RL model based on Proximal Policy Optimization (PPO) [21]. PPO performs periodic policy updates to ensure stable and efficient learning. Through iterative execution of this process, agents learn buffer reallocation strategies that minimize SLA cost under dynamic workloads.

Workload. We consider the following access patterns: *scan*, *latest*, *zipfian*, and *uniform*. A scan access pattern performs a range access to q pages (q is a parameter with a default value of 100). A latest-access pattern is one in which recently accessed pages are more likely to be re-accessed in the future. A Zipfian/uniform access pattern is one where page access follows a Zipfian/uniform distribution.

Our workload is generated by YCSB, denoted by W_1, and comprises eight tenants. Each tenant maintains 2,000,000 records, each 1 KB in size, yielding a total database size of approximately 2 GB. Half of the tenants have an SLA price of 2, with a promised buffer size of 10% of the database size, while the remaining half have an SLA price of 1, with a promised buffer size of 5% of the database size. Each tenant has aforementioned access patterns: *scan*, *latest*, *zipfian*, and *uniform*. The scan pattern consists of 100,000 operations, while the other three consist of 50,000,000 operations. To maximize workload diversity, the execution sequence of these access patterns differs across tenants with the same SLA price. The actual buffer size is set to 50% of the total promised buffer size by default.

Training. We train the RL model on a Python simulator using W_1. During training, the actual memory size was fixed at 50% of the total promised buffer size. All subsequent experiments on `OpenGauss` employed the agent pre-trained in the simulator under this configuration. The reward coefficient β is set to 0.3. Hidden layer size is $[50, 50]$. With a learning rate of 10^{-4}, the discount factor is 0.9. We train it for 1024 iterations with a batch size of 32.

Competitors. We compare our MARL against the following three buffer allocation schemes. In addition, we adopt `OpenGauss`'s default ClockSweep (an improved Clock algorithm) as the buffer replacement scheme.

- *Shared*: Use a shared buffer for all tenants.
- *Static*: The buffer size proportion of each tenant equals the ratio of their promised buffer size to the total promised buffer size.
- *SAM* [19]: An exploitation-exploration driven buffer allocation scheme.

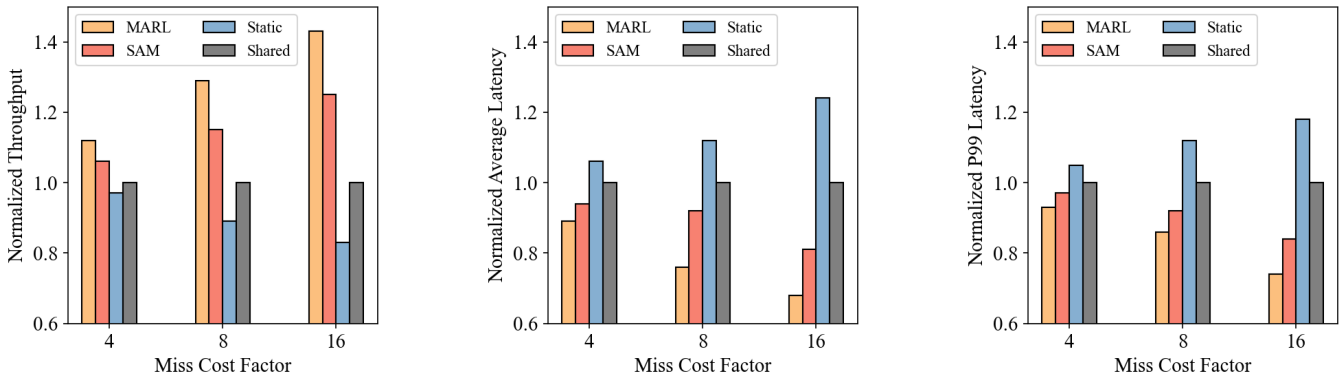
B. Experimental Results

Throughput and Latency. In real-world cloud database systems, cache misses will incur additional network access overhead, making them orders of magnitude slower than local disk accesses [22] [23]. To simulate this overhead, we add a delay into the critical path of cache misses, calculated as the measured local disk latency multiplied by a *miss_cost_factor*. We then evaluate the performance gains in throughput and latency of MARL under different *miss_cost_factors*, the actual buffer size is fixed at 50% of the total promised buffer size.

Due to MARL's advantage in miss ratio, the performance gap between MARL and other baselines widens substantially as *miss_cost_factor* increases. As shown in Fig. 2, MARL substantially outperforms the baselines as the *miss_cost_factor* increases. At a factor of 4, MARL improves throughput by 5.1% to 15.4% compared to the three baselines, while reducing average latency by 6.4% to 17.3% and P99 latency by 4.3% to 11.5%. When the factor is increased to 8, MARL yields 12.2% to 44.1% higher throughput, 17.3% to 32.2% lower average latency, and 7.3% to 23.2% lower P99 latency. At a factor of 16, MARL achieves remarkable throughput gains of 17.0% to 72.2% and reduces both average and P99 latencies by up to 45.1% and 37.4%, respectively.

Miss Ratio and SLA Cost. As shown in Fig. 3 and Fig. 4, MARL consistently achieves the lowest miss ratio and SLA cost across varying buffer pool sizes. Specifically, compared to SAM, Static, and Shared, MARL reduces the average miss ratio by 5.8%, 11.7%, and 9.7%, respectively. Similarly, it

[1]https://opengauss.org/en/



(a) Normalized Throughput (b) Normalized Average Latency (c) Normalized P99 Latency

Fig. 2: Comparison of latency and throughput under different miss cost

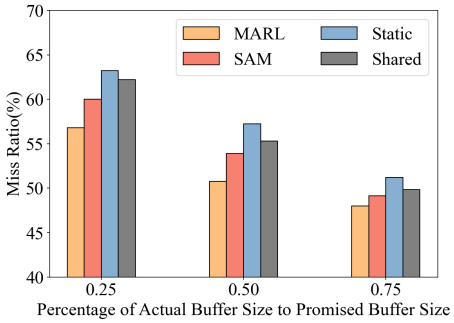


Fig. 3: Miss ratio under different buffer sizes.

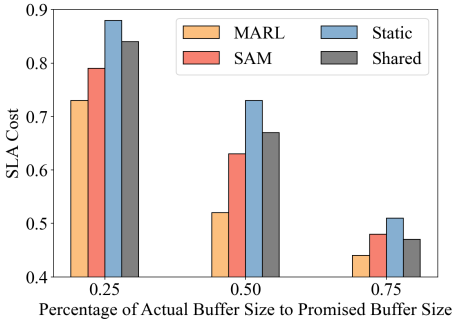


Fig. 4: SLA cost under different buffer sizes.

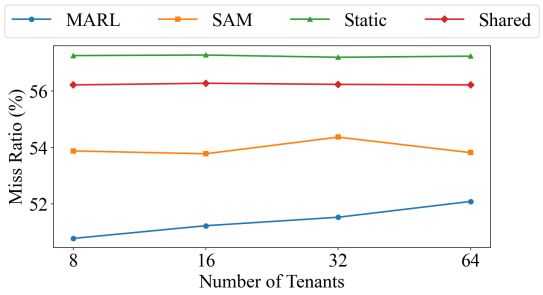


Fig. 5: Miss ratio under varying tenant numbers.

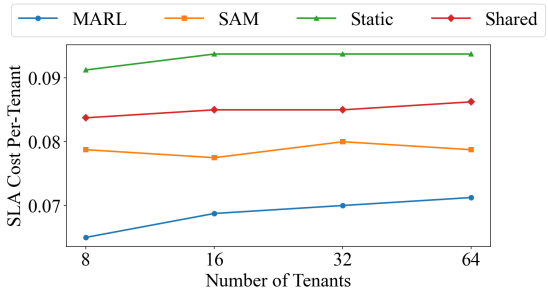
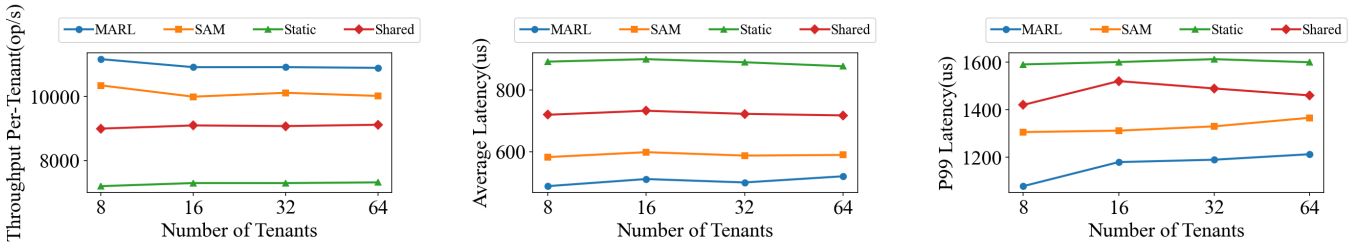


Fig. 6: SLA cost under varying tenant numbers.

reduces the average SLA cost by 15.4%, 26.3%, and 20.1% relative to these baselines.

These improvements are primarily attributed to MARL's SLA-aware multi-agent coordination mechanism, which intelligently balances resource allocation across agents to minimize SLA costs. In contrast, Static policy fails to dynamically redistribute buffer resources among tenants, resulting in significant resource waste and high SLA costs. Shared policy suffers from inter-tenant interference due to insufficient isolation for tenants with low buffer utility. Furthermore, although SAM adapts dynamically, its conservative adaptation strategy and lack of direct SLA-optimization objectives prevent it from matching MARL's performance under highly dynamic workloads.

Scalability. To evaluate scalability, we scale the baseline 8-tenant workload W_1 to 16, 32, and 64 tenants by proportionally replicating the workload instances and the buffer pool size accordingly. Fig. 5 and Fig. 6 show the miss ratios and SLA costs across different numbers of tenants, indicating that MARL achieves stable, better performance than its competitors. As shown in Fig. 7 , MARL demonstrates strong performance stability: as the number of tenants grows, changes in miss ratio, throughput, and average latency remain within 3%. Although the per-tenant average SLA cost and P99 latency show minor variations (7%–8%), MARL consistently maintains a significant advantage over the other methods.



(a) Throughput (b) Average Latency (c) P99 Latency

Fig. 7: Throughput and latency comparison under varying tenant numbers.

C. Overhead Analysis

MARL is lightweight and introduces minimal overhead to the database system. It takes 4 hours to pre-train the model offline. During deployment, the RL agent's under 64 concurrent accesses to a single tenant show that SLA measurement has no observable impact on system performance. In addition, tasks are evenly distributed across four CPU cores, and each model incurs a 20.3-millisecond fine-tuning overhead every 10 time windows (5 seconds per window) and a 0.9-millisecond inference overhead per time window. Both fine-tuning and inference are off the critical path of buffer requests. SLA measurement is similarly lightweight, since it only incurs several metadata adjustments. Evaluations of throughput and latency under 64 concurrent accesses to a single tenant show that SLA measurement has no observable impact on system performance, consistent with previous studies [24] [1].

V. Conclusions and Furture Work

In this paper, we presented a new buffer management scheme for multi-tenant databases. Based on the SLA-aware cost model, we propose using multi-agent reinforcement learning to fine-tune buffer size. We conduct extensive experiments on various multi-tenant workloads on a real database system to prove the effectiveness of our method.

Future work will focus on several aspects. First, while the current MARL framework centers on buffer pool management, expanding the optimization objective to include multi-dimensional resource coordination, such as CPU cycles, I/O bandwidth, and network throughput, would enable a more holistic approach to SLA fulfillment in complex cloud environments. Second, the research could evolve toward more decentralized and communicative architectures in which agents exchange state embeddings to proactively negotiate resource redistribution.

References

[1] V. R. Narasayya, I. Menache, and et al., "Sharing buffer pool memory in multi-tenant relational database-as-a-service," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 726–737, 2015.

[2] V. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: State-of-the-art, challenges and opportunities," in *SIGMOD*, 2022, p. 2465–2473.

[3] G. Li, H. Dong, and et al., "Cloud databases: new techniques, challenges, and opportunities," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3758–3761, 2022.

[4] H. Dong, C. Zhang, and et al., "Cloud-native databases: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 12, pp. 7772–7791, 2024.

[5] H. Lei, C. Li, and et al., "X-stor: A cloud-native nosql database service with multi-model support," *Proc. VLDB Endow.*, vol. 17, no. 12, p. 4025–4037, 2024.

[6] P. Arora, S. Chaudhuri, and et al., "Flexible resource allocation for relational database-as-a-service," *Proc. VLDB Endow.*, vol. 16, no. 13, pp. 4202–4215, 2023.

[7] H. Liu, R. Li, and et al., "Tao: Improving resource utilization while guaranteeing SLO in multi-tenant relational database-as-a-service," *Proc. ACM Manag. Data*, vol. 2, no. 4, pp. 205:1–205:26, 2024.

[8] R. Kang, Y. Chen, and et al., "ABase: the multi-tenant nosql serverless database for diverse and dynamic workloads in large-scale cloud environments," in *Companion of SIGMOD/PODS 2025*, 2025, pp. 471–484.

[9] S. Yin, F. Morvan, and et al.n, "MTD-DS: an sla-aware decision support benchmark for multi-tenant parallel dbmss," *IEEE Trans. Knowl. Data Eng.*, vol. 37, no. 5, pp. 2743–2755, 2025.

[10] F. Qazi, D. Kwak, and et al.n, "Service level agreement in cloud computing: Taxonomy, prospects, and challenges," *Internet Things*, vol. 25, p. 101126, 2024.

[11] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, low overhead replacement cache," in *FAST*, 2003. [Online]. Available: https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache

[12] Z. Song and D. S. B. andet al., "Learning relaxed belady for content distribution network caching," in *NSDI*, 2020, pp. 529–544.

[13] L. V. Rodriguez, F. Yusuf, and et al., "Learning cache replacement with CACHEUS," in *FAST*, 2021, pp. 341–354.

[14] J. Yang, Z. Mao, and et al., "GL-Cache: Group-level learning for efficient and high-performance caching," in *FAST*, 2023, pp. 115–134.

[15] Z. Song, K. Chen, and et al.i, "HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network," in *NSDI*, 2023, pp. 1149–1163.

[16] X. Hu, X. Wang, and et al., "Lama: Optimized locality-aware memory allocation for key-value cache," in *NSDI*, 2015, p. 57–69.

[17] Y. Zhang, P. Huang, and et al., "Osca: an online-model based cache allocation scheme in cloud block storage systems," in *USENIX ATC*, 2020, p. 785–798.

[18] A. Cidon, A. Eisenman, and et al., "Cliffhanger: Scaling performance cliffs in web memory caches," in *NSDI*, 2016, p. 379–392.

[19] H. Zhang, D. Zuo, and et al., "SAM: A stability-aware cache manager for multi-tenant embedded databases," *CoRR*, vol. abs/2507.22701, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.22701

[20] G. Li, X. Zhou, and et al., "opengauss: an autonomous database system," *Proc. VLDB Endow.*, vol. 14, no. 12, p. 3028–3042, 2021.

[21] Y. Chen, W. Shih, and et al., "Pairs trading strategy optimization using proximal policy optimization algorithms," in *IEEE BigComp*, 2023, pp. 40–47.

[22] X. Yang, Y. Zhang, and et al., "Polardb-scc: A cloud-native database ensuring low latency for strongly consistent reads," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3754–3767, 2023.

[23] W. Cao, Z. Liu, and et al., "Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database," *Proc. VLDB Endow.*, vol. 11, no. 12, p. 1849–1862, 2018.

[24] A. Cidon, D. Rushton, and et al., "Memshare: a dynamic multi-tenant key-value cache," in *USENIX ATC*, 2017, pp. 321–334.